ANL/ALCF-19/3

A Study of Geodesic Distance Kernel on an Integrated GPU

Argonne Leadership Computing Facility

ANL/ALCF-19/3

A Study of Geodesic Distance Kernel on an Integrated GPU

prepared by Zheming Jin

Argonne Leadership Computing Facility, Argonne National Laboratory

November 25, 2019

A Study of Geodesic Distance Kernel on an Integrated GPU

I. Introduction

Graphics processing units (GPUs) are commonly used for graphics and general-purpose (GPGPU) computation. Currently, NVIDIA's GPUs, through its CUDA framework for GPGPU programming, are ubiquitous for high-performance computing. Another class of GPUs, with a central processing unit (CPU) and a GPU integrated on the same chip, is commonly used in laptops, desktop computers, and low-power servers. While they are not designed to outperform discrete GPUs due to the power, area, and thermal constrains [1], there is a need to better understand the performance of a processor with an integrated GPU for floating-point intensive applications. The study helps us better understand the characteristics of hardware and software development tools, and the benefit of offloading such applications to an integrated GPU.

Toward that end, we choose a floating-point intensive kernel, port the kernel with a vendor-neutral framework, and evaluate the optimizations of the kernel on a server with a CPU and a GPU integrated on a chip. Specifically, we focus on the geodesic distance kernel. It has both arithmetic and trigonometric functions, and is widely used in geographical information system (GIS) for computing the distance between two locations on the earth. We develop and optimize the kernel written in Open Computing Language (OpenCL), an open standard for writing portable programs on CPUs, GPUs, and other types of hardware accelerators [2]. We show the impact of compiler's optimization options on kernel performance on an Intel® Iris™ GPU. We also compare the performance and energy consumption of the OpenCL applications executing on the processor. We summarize our key findings as follows:

- When fast relaxed math is enabled for building the single-precision floating-point intensive kernel, the compiler can take advantage of the transcendental functions implemented on the GPU. However, such functions are not available for double-precision operations. Compared to the baseline in which optimizations are disabled, the floating-point optimizations improve the performance of the single- and double-precision floating-point kernels by 9.4X and 4.8X on an Intel® GEN9 integrated GPU, respectively.
- Using a floating-point multiply-and-add OpenCL kernel to characterize the floating-point performance with respect to arithmetic intensity, we determine the arithmetic intensity required to achieve peak floating-point performance.

The report is structured as follows. Section II summarizes the compute architecture of an Intel® integrated GPU. Section III presents the optimized OpenCL kernel in details and the analysis of the compiler optimizations of the kernel. In Section IV, we show the experimental results on the GPU. Section V summarizes related studies, and Section VI concludes the report.

II. Background

While mainstream integrated GPUs are produced by ARM, Intel®, and NVIDIA, the report is focused on the architecture of an Intel® integrated GPU. This kind of GPU connects to CPU cores via a ring interconnect, and they share main memory with CPU cores. To reduce data access latency from main memory, an integrated GPU maintains a memory hierarchy comprised of register files, instruction and data caches. Some products include an embedded DRAM (eDRAM) behind last-level cache to further reduce latency to system memory for higher effective bandwidth. The building block of the graphics compute architecture is the execution unit (EU). It is a combination of simultaneous multi-threading and fine-grained interleaved multi-threading. Each EU can run seven threads concurrently to hide memory access latency. Each thread has 128 SIMD-8 32-bit registers. Each EU can co-issue to four instruction processing units (IPUs) including two floating-point units (FPUs), a branch unit for branch instructions, and a message unit for memory operations, sampler operations, and other longer-latency system communications. Each FPU supports both floating-point and integer operations, and can execute up to four 32-bit floating-point or integer operations. However, only one FPU provides support for transcendental math functions and double-precision floating-point operations.

From the perspective of an OpenCL kernel, multiple kernel instances, which are equivalent to OpenCL work-items, are executed simultaneously within a hardware thread. For a SIMD-16 compile of a kernel, 112 kernel instances can be executing concurrently on an EU. If there is a divergent branch in one or more kernel instances, an EU's branch unit keeps track of such divergence and generates masks to control which instances need to execute the branch.

Arrays of EUs are organized as a subslice. The number of EUs per subslice depend on the generation of compute architecture. Each subslice contains a thread dispatcher unit and supporting instruction caches. In addition, it includes a sampler unit for sampling texture and image surfaces, and a data port memory management unit for a variety of general-purpose buffer accesses, scatter/gather operations, as well as shared memory accesses. Subslices are grouped into slices. In general, a slice has three subslices, but the number of slices depend on products and their generations. A slice integrates additional logics for thread dispatch routing, banked L3 data cache, banked shared memory, and fixed function logic for atomics and barriers.

III. Kernel Implementation

A. Geodesic Distance Kernel in OpenCL

The geodesic distance kernel calculates the distance between two geographic coordinates on the earth's surface.

Earth's shape is modelled as an ellipsoid. The shortest distance between two points along the surface of an ellipsoid is along the geodesic. The methods for computing the geodesic distance are available in GIS, software libraries, standalone utilities, and online tools [3]. The OpenCL kernel is based on the open-source implementation [4] of the solution to the inverse geodesic problem [5].

In the Appendix, we show the N-Dimensional Range kernel in OpenCL-style pseudocode. "FP" and "FP4" represent a generic floating-point data type and its vector data type, respectively. The OpenCL vector type can pack a pair of coordinates into a vector, each of which is represented as latitude and longitude in degree. While distance calculation using double precision is preferred for accuracy, we extend it to include single precision in our study. The kernel arguments "total" and "offset" are needed for data streaming on a discrete GPU. When a GPU and a CPU are integrated, streaming is not needed to overlap communication and computation. Hence, they are equal to the global work size and zero, respectively. The kernel can be logically divided into three building blocks. The second block (lines 26 to 43) implements an iterative method to compute the geographical distance between two given points. The kernel is floating-point intensive with more than 100 arithmetic and trigonometric floating-point operations.

B. *Analysis of Floating-point Optimizations*

When building an OpenCL kernel, we can enable the OpenCL-specific optimization options to make the compiler attempt to improve the performance and reduce code size of a kernel. In this study, we focus on three optimization options for building an OpenCL kernel [6]: disable all optimizations (no-opt), default (default), and fast relaxed math (opt). We want to have a better understanding of the impact of each option on the code size and the number of floating-point arithmetic instructions at the assembly level.

We use a command-line utility ("ioc64") in Intel® SDK for OpenCL to generate an assembly file from an OpenCL kernel. The utility allows a user to specify build options for a kernel. While generating an assembly file, the utility prints a summary of device information and memory usage of a kernel. The utility can also build an OpenCL kernel offline as opposed to at runtime.

Tables I lists the impact of the build options upon the number of floating-point instructions for the single- and double-precision floating-point geodesic distance kernels targeting an integrated GPU. LOC counts the number of lines (including labels) in the assembly file generated from the OpenCL kernel. Compared to the kernel with the optimizations disabled or with the default option, relaxing the single-precision floating-point operations can reduce the LOC by approximately 10X and more. We attribute the significant reduction in code size to the instantiation of three trigonometric functions for the sine and cosine operators in the kernel. These extended math functions take advantage of the extended math capability provided by the underlying FPU. In addition, all divide instructions (div) are replaced with the inverse instructions (inv) to reduce the overhead of floating-point divide.

On the other hand, relaxing the double-precision floating-point operations can reduce the LOC by at most 1.6X. The result suggests that there are no special processing units for double-precision floating-point trigonometric functions in an FPU. For both precisions, the divide instructions are replaced with the inverse instructions to reduce the overhead of floating-point divide. In addition, the compiler is able to use the built-in "mad" and "rsqrt" functions for efficient computations. Overall, the compiler is better at optimizing a

TABLE I. IMPACT OF THE OPTIMIZATIONS OF THE GEODESIC DISTANCE KERNEL ON THE NUMBER OF INSTRUCTIONS IN THE ASSEMBLY

Name	Note	Float32 no-opt.	Float32 default	Float32 opt.	Float64 no-opt.	Float64 default	Float64 opt.
LOC	Lines of code	2144	1657	145	5969	4519	3837
mul	Multiplication	218	220	41	530	484	486
add	Addition	381	258	15	1275	859	474
mad	Multiply-and-add	196	201	26	963	908	679
div	Division	11	8	0	0	0	0
inv	Inverse	2	2	8	32	26	22
sqrt	Square root	2	2	2	0	0	0
rsqrt	Reverse square root	2	2	3	8	8	8
sin	Sine	0	0	3	0	0	0
cos	Cosine	0	0	3	0	0	0
cmp	Compare	108	89	3	89	86	79
sel	Select	186	194	4	34	26	35
mov	Move	311	160	24	1480	957	917
shl	Shift left	89	86	2	235	124	124
shr	Shift right	138	96	0	138	132	132
or	Logical OR	121	120	1	132	125	125
and	Logical AND	73	68	0	237	203	191

single-precision kernel than a double-precision kernel.

IV. Experiment

A. Setup

In our experiment, the server has an Intel® Xeon® E5-1585 v5 CPU running at 3.5 GHz. The CPU has four cores and each core has two threads. The theoretical memory bandwidth is 34.1 GB/s. The integrated GPU is Skylake GT3e, Generation 9.0. The specifications for the GPU are summarized in Tables II. The official thermal power draw of the GPU is not available. For the GPU compute runtime [7], the device version is OpenCL 2.1 NEO and the driver version 19.43.14583. The maximum work-group size (local size) is 256. Empirical results show that the runtime can select an appropriate work-group size; therefore we have the OpenCL implementation determine how to break the global work-items into appropriate work-group instances. We use Red Hat Enterprise Linux 7 (version 7.6) and the kernel is 5.3.1. We build the OpenCL application with the g++ compiler, version 4.8.5.

To minimize data transfers between the integrated GPU and the CPU, we make use of the zero-copy buffers which can reduce the communication cost and energy consumption. We measure the total elapsed time of executing an OpenCL application on a target device. This includes the initialization of OpenCL runtime, offloading the kernel to the GPU, and reading back the GPU results. We use “ioc64” to build the OpenCL kernel offline as opposed to at runtime, and choose the minimum execution time of 32 runs.

The input coordinates are retrieved from Maxmind’s world cities database that includes city, region, country, latitude, and longitude. In our experiment, we extract 2^{21} cities with unique locations around the world, and choose six cities in five continents (Bombay, Melbourne, Waltham, Moscow, Glasgow, and Morocco) from which the kernel computes distances to each of 2^{21} cities.

B. Performance Characterization

Two key attributes of computer graphics computation are

```
kernel void multiply_add (
  global const TYPE *restrict a,
  global       TYPE *restrict c )
{
  int i = get_global_id(0);
  TYPE sum = 0;
  for (int j = 0; j < N; j++)
  {
    sum += a[i] * j;
  }
  c[i] = sum;
}
```

Listing 1. The OpenCL reduction kernel with the parameters “TYPE” and “N” set at compile time

data parallelism and independence, which can be combined in a single concept known as arithmetic intensity [8]. In this study, we choose a floating-point multiply-and-add kernel as shown in Listing 1 to evaluate the number of floating-point operations performed by the kernel relative to the amount of memory accesses that are required to support those operations. It should be pointed out that we use this kernel to *estimate* the performance of the floating-point operations with respect to the arithmetic intensity on the GPU. Our focus is not the maximum floating-point performance that can be achieved on the GPU. The global work size is 2^{27}, and the work-group size is set by the OpenCL runtime. Each work-item iterates over a “for” loop to accumulate the sum of the product of the content of the array element at index “i” and the value of the loop index. The parameters “TYPE” and “N” in the OpenCL kernel are set at compile time. From the OpenCL kernel perspective, there are two single-precision floating-point operations for each work-item when “N” equals one and “TYPE” is “float”.

Figures 1 and 2 show the giga floating-point operations per second (GFLOPS) with respect to arithmetic intensity for the single- and double-precision floating-point multiply-and-

TABLE II. THE GPU INFORMATION

Parameter	Iris™ Pro Graphics P580
Technology	14 nm
Base Freq.	0.35 GHz
Max Dynamic Freq.	1.15 GHz
Embedded DRAM	128 MB
Slices/Subslices	3/9
Execution Units	72
Max Memory Size	64 GB
Thermal Power Draw	N/A

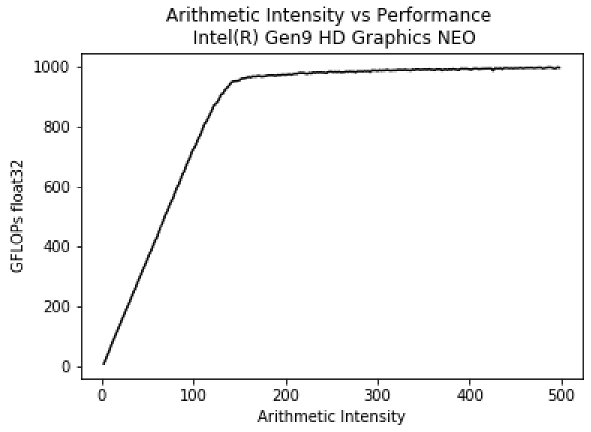

Fig 1. Giga floating-point operations per second (GFLOPS) with respect to arithmetic intensity for the single-precision (float32) floating-point multiply-and-add kernel

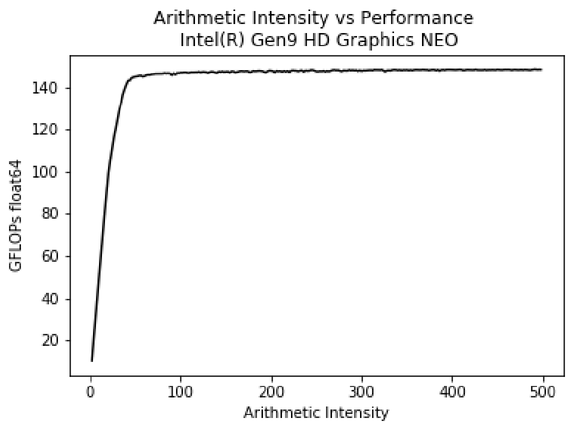

Fig 2. Giga floating-point operations per second (GFLOPS) with respect to arithmetic intensity for the double-precision (float64) floating-point multiply-and-add kernel

add kernel on the GPU, respectively. The GFLOPS levels off at approximately 1000 and 150 for the single- and double-precision kernels, respectively. The "peak" single-precision GFLOPS is more than five times higher than the "peak" double-precision GFLOPS when the arithmetic intensity is above 150. The performance trends show that achieving the "peak" GFLOPS on the GPU requires a minimum arithmetic intensity of 50.

C. *Performance Comparison*

The relationship between the GFLOPS and arithmetic intensity indicates that it is beneficial to offload a floating-point intensive kernel to the GPU. Ultimately, we need to know the raw performance of the kernel implementations.

Table III shows the elapsed time in millisecond (ms) on the GPU with respect to the three optimization options, and the performance speedup in terms of execution time (Speedup^T) and precision (Speedup^P). The single-precision kernel is only about 2.6X faster than the double-precision kernel when all optimizations are disabled. However, the single-precision floating-point optimizations can achieve 5X speedup over the optimized double-precision floating-point operations. On the other hand, the floating-point optimizations reduce the baseline execution time by 9.4X and 4.8X for the single- and double-precision kernels, respectively. The performance improvement shows that it is important to apply the floating-point optimizations whenever possible. The comparison also suggests that more double-precision floating-point optimizations at the hardware level

TABLE III. MINIMUM EXECUTION TIME (MILLISECOND) OF 32 RUNS ON THE GPU. THE GLOBAL WORK SIZE IS 2^{21}.

Precision	Time No-opt.	Time Default	Time Opt.	Speedup^T
Float32	113	38	12	9.4
Float64	290	184	60	4.8
Speedup^P	2.6	4.8	5	

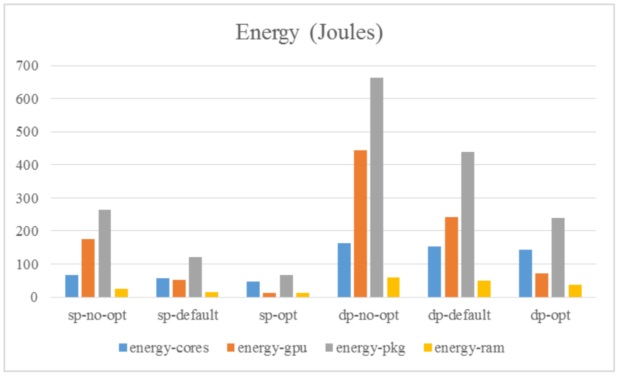

Fig. 3. The energy consumption (cores, GPU, package, DRAM) of running the OpenCL application with the un-optimized, default, and optimized kernels executing on the GPU.

are needed for high-performance computing applications which require double-precision floating-point operations for accuracy and stability.

We look at GPU kernel execution per code line with the GPU In-kernel Profiling in Intel® VTune™ Amplifier [9]. The profiling tool can estimate the GFLOPS of a kernel via GT-Pin [10]. When the floating-point optimizations are enabled, the reported GFLOPS is 422 and 79 for the single- and double-precision floating-point kernels, respectively. Based on the specification in Table II, the theoretical maximum GFLOPS of the GPU is 1324.8 for single precision and 331.2 for double precision. Hence, the kernel can achieve 32% and 24% of the theoretical single- and double-precision floating-point performance, respectively. Compared to the "peak" performance measured using the multiply-and-add kernel, the kernel can achieve 43% and 53% of the "peak" single- and double-precision GFLOPS, respectively. We attribute low performance efficiency to the overhead of executing complex trigonometric functions in an iterative loop.

D. *Energy Consumption*

We measure the energy consumption of running the OpenCL applications using Linux's "perf" interface [11]. Figure 3 shows the energy in Joules of the event types "cores", "gpu", "pkg", and "ram" when running the OpenCL application in single- and double-precisions on the processor. For a compute-bound application, we expect that the DRAM energy consumption is the lowest. Using the floating-point optimizations reduce the package energy by 3.9X and 2.8X for the single- and double-precisions, respectively. On the other hand, the double-precision floating-point operations consume about 3.1X, 5.5X, 3.6X, and 2.8X more energy than that of the single-precision floating-point operations for the four energy events, respectively.

V. RELATED WORK

In [1], the authors presented the architectures of Skylake® and Kabylake® integrated GPUs, and characterized the performance of the two GPUs through a collection of micro-benchmarks. Although it is the first work that takes a detailed look at Intel® integrated GPU architectures, the paper does not provide the insight of floating-point performance with respect to arithmetic intensity. Compared to the extensive studies on NVIDIA's discrete GPUs, there are limited studies on Intel's integrated GPUs [12, 13, 14, 15 , 16] due to the limitations in performance and programming framework. Compared to the previous work, we add the analysis of compiler optimizations, the characterization of floating-point performance, and the performance evaluation of the single- and double-precision floating-point implementations on the GPU.

VI. Conclusion

We use the geodesic distance kernel in OpenCL as a case study to better understand the performance of a floating-point intensive kernel on a processor with a CPU and a GPU integrated on the same chip. Although an integrated GPU is not designed to outperform a discrete GPU, our experimental results show that the floating-point optimizations can improve the raw performance of the single- and double-precision floating-point kernels by 9.4X and 4.8X on the GPU. In the meantime, it can reduce the package energy by 2.8X for the single precision, and 3.9X for the double precision. The GFLOPS with respect to arithmetic intensity shows that the arithmetic intensity plays an important role in improving the number of floating-point operations per second. Because of the hardware architecture of an FPU, the compiler can generate efficient instructions for the single-precision transcendental functions in the kernel whereas there are no special function units for the double-precision floating operations. We expect that technological advances will improve the performance of double-precision floating-point operations from the hardware level.

Acknowledgment

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] Gera, P., Kim, H., Kim, H., Hong, S., George, V. and Luk, C.K.C., 2018, April. Performance Characterisation and Simulation of Intel's Integrated GPU Architecture. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (pp. 139-148). IEEE.

[2] Stone, J.E., Gohara, D. and Shi, G., 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering, 12(3), p.66.

[3] Wikipedia webpage, https://en.wikipedia.org/wiki/Geographical_distance

[4] GpsDrive Homepage, http://www.gpsdrive.de/

[5] Geographiclib Homepage, https://geographiclib.sourceforge.io/2009-03/geodesic.html

[6] https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clBuildProgram.html

[7] https://github.com/intel/compute-runtime

[8] Harris, M., 2005, July. Mapping computational concepts to GPUs. In ACM SIGGRAPH 2005 Courses (p. 50). ACM

[9] https://software.intel.com/en-us/vtune-amplifier-help-gpu-in-kernel-profiling

[10] Kambadur, M., Hong, S., Cabral, J., Patil, H., Luk, C.K., Sajid, S. and Kim, M.A., 2015, October. Fast computational gpu design with gt-pin. In 2015 IEEE International Symposium on Workload Characterization (pp. 76-86). IEEE.

[11] Jim Jeffers et al.: Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, Morgan Kaufmann Publishers (2016)

[12] Lupescu, G., Slusanschi, E.I. and Tapus, N., 2016, September. Analysis of OpenCL work-group reduce for Intel GPUs. In 2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)(pp. 417-423). IEEE.

[13] Arvid, J., 2017. Analysis of GPU accelerated OpenCL applications on the Intel HD 4600 GPU.

[14] Tseng, J., Wang, R., Tsai, J., Edupuganti, S., Min, A.W., Woo, S., Junkins, S. and Tai, T.Y.C., 2016, June. Exploiting integrated GPUs for network packet processing workloads. In 2016 IEEE NetSoft Conference and Workshops (NetSoft) (pp. 161-165). IEEE.

[15] Carroll, M.R., 2018, May. Improving Performance of OpenCL Workloads on Intel Processors with Profiling Tools. In IWOCL(pp. 6-1).

[16] https://software.intel.com/en-us/blogs/2016/10/05/micro49-tutorial-on-intel-processor-graphics-microarchitecture-and-isa

APPENDIX GEODESIC DISTANCE KERNEL IN OPENCL

```
kernel void
gd (global FP4 *restrict coordinate,
    global FP  *restrict distance,
    const uint           total,
    const uint           offset )
{
  i = get_global_id(0) + offset ;
  if ( i >= total ) return ;
  lat1 = coordinate[i].s0 ;
  lon1 = coordinate[i].s1 ;
  lat2 = coordinate[i].s2 ;
  lon2 = coordinate[i].s3 ;
  rad_lon1 = lon1[i] * TO_RADIAN ;
  rad_lat1 = lat1[i] * TO_RADIAN ;
  rad_lon2 = lon2[i] * TO_RADIAN ;
  rad_lat2 = lat2[i] * TO_RADIAN ;
  tu1 = COMPRESSION_FACTOR * sin ( rad_lat1 ) / cos ( rad_lat1 ) ;
  tu2 = COMPRESSION_FACTOR * sin ( rad_lat2 ) / cos ( rad_lat2 ) ;
  cu1 = 1.0 / sqrt ( tu1 * tu1 + 1.0 ) ;
  su1 = cu1 * tu1 ;
  cu2 = 1.0 / sqrt ( tu2 * tu2 + 1.0 ) ;
  s = cu1 * cu2 ;
  baz = s * tu2 ;
  faz = baz * tu1 ;
  x = rad_lon2 - rad_lon1 ;
  do {
    sx = sin ( x ) ;
    cx = cos ( x ) ;
    tu1 = cu2 * sx ;
    tu2 = baz - su1 * cu2 * cx ;
    sy = sqrt ( tu1 * tu1 + tu2 * tu2 ) ;
    cy = s * cx + faz ;
    y = atan2 ( sy, cy ) ;
    sa = s * sx / sy ;
    c2a = - sa * sa + 1.0;
    cz = faz + faz ;
    if ( c2a > 0.0 ) cz = -cz / c2a + cy ;
    e = cz * cz * 2.0 - 1.0 ;
    c = ( ( -3.0 * c2a + 4.0 ) * FLATTENING + 4.0 ) * c2a * FLATTENING / 16.0 ;
    d = x ;
    x = ( ( e * cy * c + cz ) * sy * c + y ) * sa ;
    x = ( 1.0 - c ) * x * FLATTENING + rad_lon2 - rad_lon1 ;
  } while ( fabs ( d - x ) > EPS ) ;
  x = sqrt( ELLIPSOIDAL * c2a + 1.0 ) + 1.0 ;
  x = ( x - 2.0 ) / x ;
  c = 1.0 - x ;
  c = ( x * x / 4.0 + 1.0 ) / c ;
  d = ( 0.375 * x * x - 1.0 ) * x ;
  x = e * cy ;
  s = 1.0 - e - e ;
  s = ( ( ( ( sy * sy * 4.0 - 3.0 ) * s * cz * d / 6.0 - x ) *
             d / 4.0 + cz ) * sy * d + y ) * c * POLAR_RADIUS ;
  distance[i] = s ;
}
```